VIRTUAL FILE SYSETM INTERFACE TO CONFIGURATION DATA OF A PLD

FIELD OF THE INVENTION

[0001] The present invention generally relates to accessing configuration data of a programmable logic device (PLD).

BACKGROUND

[0002] The technology of programmable logic devices (PLDs) has progressed to a level such that it is possible to implement a microprocessor on a PLD and host an operating system such as Linux. Not only may this support various system on a chip (SOC) applications, but with the capabilities of a PLD such as a Virtex field programmable gate array (FPGA) from Xilinx, the SOC is reconfigurable to suit changing requirements or adjust to changing environmental conditions.

[0003] The configuration of a PLD such as an FPGA, generally refers to the state of the configurable logic resources of the device as programmed to accomplish a set of required functions. Data that is loaded into the device for a desired configuration may be referred to as configuration data. The state of the device generally refers to the contents of registers in the device at a chosen time while the device is performing the desired functions.

[0004] Devices such as the Virtex FPGA have various modes available for accessing the configuration data and state. For example, for general needs the SelectMap interface of the Virtex FPGA may be used to load a configuration bitstream into the device from an external memory or read back configuration data from the device. For testing and diagnostic activities, a boundary scan interface may be used to establish or read back state data. The internal configuration access port (ICAP) may be implemented on a Virtex FPGA to manipulate configurable resources by, for

example, an embedded processor, (i.e., self-controlled configuration) after the device has already been configured.

[0005] Even though these interfaces for accessing configuration data or device state are suitable for most purposes, the interfaces are low-level (less abstract) and may be unnecessarily complex for some applications. For example, to the developer of an application in which the PLD is to be re-configured while being part of a running system (run-time reconfiguration), the low-level interface may not be familiar. The developer's lack of familiarity may present scheduling, reliability, and cost issues.

[0006] The present invention may address one or more of the above issues.

SUMMARY OF THE INVENTION

[0007] The various embodiments of the invention provide methods and apparatus that support access to data in a programmable logic device (PLD). A hierarchy of directories and files are maintained in a virtual file system, which is registered with an operating system. The directories and files are associated with resources of a PLD. In response to program calls to file system routines that reference files associated with resources of the PLD, the virtual file system is invoked, and the virtual file system accesses state information in resources of the PLD. In various other embodiments, the PLD configuration data may be remotely viewed via a network file system interface.

[0008] Various means are described for providing access to PLD configuration data. Means are described for maintaining a hierarchy of directories and files in a virtual file system. In support of providing access to the PLD configuration data, means are described for providing access to PLD resources, via both local access and network access to the PLD.

[0009] It will be appreciated that various other embodiments are set forth in the Detailed Description and Claims which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] Various aspects and advantages of the invention will become apparent upon review of the following detailed description and upon reference to the drawings in which:

[0011] FIG. 1 is a block diagram that illustrates a programmable logic device (PLD) having implemented therein a microprocessor;

[0012] FIG. 2 is a functional block diagram that illustrates a system in which the configuration of a PLD is made accessible via a virtual file system;

[0013] FIGs. 3A, 3B, 3C, and 3D illustrate examples of various alternative hierarchies of directories and files that are associated with configuration data of an FPGA;

[0014] FIG. 4 is a flowchart of an example process for establishing a hierarchical directory and file view of PLD resources in accordance with various embodiments of the invention;

[0015] FIG. 5 is a flowchart of an example process for providing read access to data of a PLD via the hierarchical directory and file view of the PLD resources; and

[0016] FIG. 6 is a flowchart of an example process for providing write access to data of a PLD via the hierarchical directory and file view of the PLD resources.

DETAILED DESCRIPTION

[0017] The various embodiments of the invention support access to PLD configuration data via program calls to file system routines of an operating system kernel. A processor is implemented on a PLD, and an operating system having program callable file system routines executes on the processor. The configuration data is managed in a virtual

file system which is registered with the operating system. Access to the PLD configuration data is made through the program invoking standard system calls for file system routines, which in turn invoke the virtual file system. The virtual file system (VFS) reads, writes, or modifies the configuration data according to the file system calls. In one embodiment, the virtual file system interfaces with a configuration controller implemented on the PLD, and the configuration controller manipulates the configuration data on the PLD.

[0018] The approaches described herein for providing access to PLD configuration data may be useful for a variety of purposes such as use of the VFS by application level archiving and revision control program suites. For example, the VFS approach may be useful for configuration management. By using the file system approach, standard revision control systems may be used to manage the configuration data for a variety of PLD applications. Revision control systems are able to reconcile the version (revision) of a given file to that in a central repository. With past approaches, revision control systems have been generally used to manage the configuration data of a PLD as a monolithic data set, i.e., the configuration bitstream. With the VFS approach, the revision control system may control and track the configuration data to a user-specified level of file abstraction.

[0019] The various embodiments of the invention support creation and restoration of archive collections of VFS files using standard file archiving tools. The data within the files represents the configuration (and/or state) of the PLD and may be applied at a remote system by using the same archive tool to re-write the files into the appropriate regions of the VFS hierarchy.

[0020] The configuration data is stored in a hierarchical directory structure, which provides device independence/abstraction. The hierarchical directory

structure may be used to allow configuration data intended for one device size/architecture/family to be applied to another device with a different size/architecture/family. This may be implemented at the VFS driver level.

[0021] FIG. 1 is a block diagram that illustrates a programmable logic device (PLD) 102 having implemented therein a microprocessor 104. The microprocessor 104 may be implemented with the PLD logic and in one embodiment hosts an operating system, such as Linux, that supports a VFS.

[0022] In addition to the microprocessor, the PLD 102 has implemented therein a configuration controller 106 which interfaces with the microprocessor via bus 108. The configuration controller supports access to the configuration state of the PLD over general interconnect circuitry within the device. For example, in one embodiment, the PLD 102 is a field programmable gate array (FPGA) such as a Virtex FPGA from Xilinx, Inc., and the configuration controller 106 may be implemented with the internal configuration access port (ICAP).

[0023] ICAP allows configuration of the internal logic resources after the device is configured. The ICAP block exists in the lower right corner of the array and emulates the SelectMAP configuration protocol from Xilinx. ICAP supports reconfiguration of the device and bus interfaces such as the peripheral component interface (PCI).

[0024] In other embodiments, the configuration controller 106 may be implemented with a SelectMap or JTAG boundary scan or other physical port having access to a configuration bitstream stored internal to the PLD, external to the PLD or any combination thereof. In addition the microprocessor hosting the operating system supporting the VFS may be internal or external to the PLD. For example, the VFS can be running on a processor external to the PLD and the VFS uses the SelectMap/BoundaryScan/other-physical-port to manipulate the configuration data on the PLD.

[0025] Depending on application and implementation requirements, the PLD may be configured with additional controllers, for example, controllers 112 and 114, that provide access to off-chip resources. For example, controller 112 may be a memory controller that provides access to memory 116. It will be appreciated that a variety of commercially available or proprietary memory architectures and interfaces may be used to implement the memory controller and memory. Other types of controllers, illustrated by block 114, may provide network or communications interfaces and access to retentive storage systems such as tape drives and disk drives.

[0026] The various embodiments of the invention provide access to configuration data of a PLD via a virtual file system (VFS). An operating system such as Linux supports virtual file systems (VFSs). For example, the Linux "/proc" VFS allows user-space applications to access the selected ones of the operating system kernel's data structures via basic file I/O operations or function calls. From the perspective of a user-space application, access is made to the configuration data by calling the same file system routines as are called for accessing a file on a storage media such as a disk. However, the access is actually to a data structure in the kernel's memory. In various embodiments of the present invention, access is to the actual configuration data of configurable logic resources of an FPGA or state data of memory cells in the FPGA. It may in general be desirable to access the FPGA configuration data and state data directly rather than caching the data in kernel memory.

[0027] FIG. 2 is a functional block diagram that illustrates a system 200 in which the configuration of PLD resources is made accessible via a virtual file system. The application need not directly interface with the configuration controller driver. Instead, the application may make file system calls to the VFS extension, which hides

the complexity of the configuration controller from the application.

[0028] In an example embodiment, an application program 202 makes file system calls 204 that reference the VFS-implemented view of the PLD resources 206. The PLD resources are represented in the VFS by the LINUX /proc file system in an example embodiment. The application view of the representation is shown as the directory hierarchy 208.

[0029] The example directory hierarchy 208 shows an FPGA directory along with example bus and kernel-memory (kmem) directories. The bus and kmem directories illustrate additional kernel-managed resources that may be accessible via a VFS. An FPGA is used as an example of a PLD whose configuration data may be made accessible via the VFS. The configuration data of the FPGA is organized into directories that represent the data by application, by platform areas, and by resources, as shown in more detail in FIGs. 3A-3D. It will be appreciated that the application subdirectory under the FPGA directory relates to the application implemented on the FPGA as compared to the application 202 which may manipulate the FPGA configuration data.

[0030] The VFS 212 operates in kernel memory space 214 and provides access to kernel data structures 216, 218, 220, and 222 for application 202, which operates in user memory space 224. The VFS is accessible to the application via file system calls, and the VFS interfaces with the respective kernel components or drivers that manage the referenced data. Example components and drivers in the kernel include a scheduling component 232, a virtual memory management component 234, and other drivers 236 for interfacing with other data structures. The other drivers may include, for example, drivers for interfacing with data communications network 238.

[0031] In one embodiment, driver 240 provides VFS access to the configuration controller 106 (FIG. 1). The data accessed by driver 240 is the configuration state of the FPGA

as referenced by the parameters in the file system calls made by application 202.

[0032] In another embodiment, the driver 240 may be invoked from a network-connected node 242 for controlling or monitoring the FPGA configuration state. This may allow a remote system to maintain a higher-level directory structure that represents the configuration data for all the PLDs that are deployed and managed by the node. Opening a directory/file at node 242 would transparently invoke network file system protocols within node 242 and the file system implemented on the PLD would act as a client. An example use of the network-connected embodiment may involve determining differences between the configuration data of separate PLDs in order to diagnose problems. Standard tools may be used with the embodiments for revision control and archiving of configuration data.

[0033] It will be appreciated that in yet another embodiment, the network-connected node 242 could implement the VFS view of directories and files of configuration data instead of the PLD implementing the file system logic.

[0034] FIGs. 3A, 3B, 3C, and 3D illustrate various alternative hierarchies of directories and files that are associated with configuration data of an FPGA. FIG. 3A shows configurable resources of the FPGA for which the configuration data may be modeled as files in a directory hierarchy. The resources directory includes sub-directories for a digital clock manager ("dcm"), FPGA flip flops ("flops"), internal configuration access port ("icap"), input/output blocks ("iobs"), instruction set processors ("isps"), boundary scan architecture ("jtag"), look-up tables ("luts"), and routing resources ("routes"). The class of configuration data is modeled as directories 252.

[0035] Within each of directories 252 are further directories or files that represent the configuration data of the FPGA for the resources identified by the directory. For example, the routes directory contains directories 254 for

routing resources of the FPGA, including "long lines", "hex lines", "double lines", and "direct-connect lines" directories. Each of directories 254 may contain a directory or file for each instance of the specific routing resource.

[0036] The platform view shown in FIG. 3B illustrates modeling of FPGA configuration data by region of the device. The example includes directories for four example regions, region1, region2, region3, and region4. In addition, the platform directory includes a directory named, interconnect, for routing resources that connect the regions. Under each of the region directories, region1, region2, region3, and region4, a file represents the configuration data for resources within an area of the FPGA associated with the directory. For example, the device may be viewed as having resources in 4 quadrants, each region directory may be a associated with one of the quadrants.

[0037] In an example embodiment, the files within each region may also be used to store statistical information pertaining to the region. For example, a file may contain information that indicates the spatial utilization of the region. The total area of the region is communicated to the VFS device driver in a configuration file or by kernel module options. The configuration file may be communicated to the VFS device driver by way of a symbolic ("sym") file handle. A sym file handle is a portal to an interpreter implemented in the VFS driver to which the configuration file may be written. In response, the VFS takes note of the information contained in the configuration file and applies whatever corresponding actions are necessary. Sym files may also be used to communicate which regions exist, the names of the cores, where the registers are placed, and the symbolic names of the registers, for example. This allows the VFS to establish a directory hierarchy for an application with the appropriate directory names and files within those directories.

[0038] The spatial utilization of the region may be determined by the VFS module monitoring which parts of the configuration data are affected during configuration. In another embodiment, the configuration data may be accompanied by a configuration mask that could be parsed to reveal the same utilization information. The files could also be used as portals through which new configuration information could be placed into the region. Furthermore, if two regions were compatible at a physical level, the file from one region directory could be copied to the other region directory, resulting in a partial configuration bitstream being loaded into the region of the FPGA associated with the target directory.

[0039] In yet another embodiment, the configuration data associated with a region directory may be further organized by configuration data and state data as shown in FIG. 3C. Configuration data refers to the state of resources of the FPGA that are programmable to implement a desired function, and state data refers to the state of elements of the FPGA that may be subject to change while the FPGA is performing its programmed functions, such as flip flops, registers, and memory. Each region directory may have separate directories for each of the configuration and state data, as illustrated by directories 262 under the directory, region1. It will be appreciated that each region directory may have configuration and state files instead of further subdirectories (entries 262 could be files instead of subdirectories).

[0040] FIG. 3D illustrates example core and subsystem subdirectories 272 under the application directory. Under the application view, there are subdirectories for the constituent cores and subsystems implemented on the FPGA. For example, the application directory in FIG. 3D includes core directories, core1 and core4, along with a subsystem directory, subsystem1, which further includes sub directories, core2 and core3. The files under each core or subsystem directory may represent the registers in the core.

Each file representing a register could be interrogated by opening and reading from the file or modified by writing to the file. The behavior of a core may be modified, to the extent allowed by the core, by reprogramming the registers via the file system calls to update the necessary file(s).

[0041] In another embodiment, each core directory may further include subdirectories that map to the levels in the application hierarchy (such as that created with a design tool). A file representing the configuration of a part of the core's design hierarchy could be moved or copied to the corresponding level in the core's directory within the VFS directory hierarchy. The file acts as a configuration portal, and overwriting the file substitutes that part of the application design hierarchy with alternative functionality. It will be appreciated that the configuration data that implements the alternative functionality must fit within the same physical region of the FPGA as that region in which the original functionality is implemented.

[0042] In support of manipulating configuration data under the application view, symbol data is provided to the driver of the configuration controller. The symbol data enumerates the core's name, type, constituent registers and physical locations, along with design information that describes the logical and physical hierarchy of the design. This information may be provided to the driver via a file in the application directory when the application 202 (FIG. 2) begins executing, and the driver populates its internal data structures with this information.

[0043] FIG. 4 is a flowchart of an example process for establishing a hierarchical directory and file view of PLD resources in accordance with various embodiments of the invention. The process begins with registration of modules that extend the VFS with the operating system kernel (step 402). Registration informs the kernel that a module exists and may be loaded when the module is to be used and unloaded when use of the module is complete, such as for dynamically

linked libraries (DLLs). The particular mechanism by which modules register with an operating system may vary for different types of operating systems. Alternatively, a module may be statically compiled with the kernel.

[0044] An application may then make file system calls to establish a hierarchy of directories and files that represent the configuration data of the PLD (step 404). In another embodiment of the VFS, the hierarchy of directories is created by the VFS module when the module is first loaded. As noted below, in the dynamic version of the VFS, the files and directories are created by the VFS in response to the application doing a 'cd'/'ls', etc. in the VFS directory space. The file system routines in turn call the VFS extensions, which create the directories and files. In making the file system calls, configurable resources of the PLD are associated with the directories and files (step 406).

[0045] File system access control and file locking may also be implemented for the directories and files that represent the configuration data of the PLD. This functionality is typically provided by the kernel system code rather than the VFS extensions. In one embodiment, the configuration controller driver 240 (FIG. 2) may set owner and access permissions for given files and directories (step 408).

[0046] It will be appreciated that the actions of the process of FIG. 4 may be performed somewhat differently when the concepts of the present invention are implemented on different operating systems.

[0047] In an alternative embodiment, the directory and files may be created on demand. That is, parts of the directory hierarchy are generated only when needed. For example, in a VFS, directories and files may be generated as a process begins execution and deleted when the process exits. In one embodiment, when an application changes a view to a particular directory (e.g., the "cd" command in LINUX) the VFS dynamically creates the file system data structures

to represent the required resources. In one embodiment, the required resources are hard coded into the implementation of the VFS module. In another embodiment, a configuration mask file may be used to only create the subset of files representing the resources used by the current FPGA configuration.

[0048] FIG. 5 is a flowchart of an example process for providing read access to data of a PLD via the hierarchical directory and file view of the PLD resources. To read configuration data, an application calls a standard file system routine to read a designated file (step 502). The file system in turn calls the VFS extension module for reading configuration data (step 504).

[0049] The extension module checks whether the application has permission to read the specified file (or directory) (step 506). If the application lacks permission, control is returned to the calling application with a code that indicates access was denied.

[0050] If the application has the requisite permission, the VFS extension modules determines from which configurable resources of the PLD the configuration data is to be read (step 510). This information may be obtained from the configuration controller driver data structures 220 (FIG. 2). The configuration controller driver is then called to read configuration data from the appropriate resources of the PLD (step 512), and the data is returned from the VFS extension to the calling application (step 514).

[0051] FIG. 6 is a flowchart of an example process for providing write access to data of a PLD via the hierarchical directory and file view of the PLD resources. The write process of FIG. 6 is similar to the read process of FIG. 5, with the difference being that a write operation is performed instead of a read operation.

[0052] To write configuration data, an application calls a standard file system routine to write a designated file (step 602). The file system in turn calls the VFS extension module

for writing configuration data (step 604). The extension module checks whether the application has permission to write the specified file (or directory) (step 606). If the application lacks permission, control is returned to the calling application with a code that indicates access was denied (step 608). If the application has the requisite permission, the VFS extension modules determine from which configurable resources of the PLD the configuration data is to be written to (step 612). The configuration controller driver is then called to write configuration data to the selected resources of the PLD (step 614).

[0053] For both the read and write processes, access to the configuration data may be controlled by way of execute permissions that are associated with executable files. For example, the configuration controller driver 240 may provide customized, executable files that manipulate the other data files available in the subdirectory tree. The executable files may be shell scripts, binary code, or the opening of a file with execute permission may be intercepted and interpreted as a request to perform the operations specified by the file.

[0054] In yet another embodiment, the contents of an executable file may be associated with redirecting control flow into the VFS module code itself. The module then performs the specific behavior represented by the given file in response to the user-space request to open the file for execution. The VFS, internally, may schedule the execution of some specific method defined within the VFS driver code. The invoked executable file immediately returns control to the caller. This mechanism is similar to invoking wrapped system calls except actual operating system calls (e.g., UNIX sys or files system calls) are not being invoked. The methods are specific to the VFS's control of the FPGA.

[0055] Still further file-based operations may be used in manipulating PLD configuration data. For example, operations on special file types, such as named and anonymous pipes, may

be used to trigger functions of the configuration controller driver 240. One example operation is the creation of a named pipe. The actions performed by the VFS extension modules depend on the part of the directory structure (e.g., 208) in which the pipe is created. For example, the VFS extensions may alter the configuration of the portion of the PLD associated with the interconnect file in the platform subdirectory when a named pipe is established between the two regions. Symbolic link files may be interpreted by the VFS extensions for a similar purpose. The difference between named pipes and symbolic link files is that the operating system may create a process for a named pipe, where as the symbolic link is a static object in the file system.

[0056] Additional system routines may be defined to operate on the files that represent PLD configuration data. An example is a routine that freezes or halts the PLD. This routine may be needed because it may take hundreds of system clock cycles to read back data from a region of PLD. During this time the states of flip flops, registers, and look-up table RAM may be changing. Thus, a halt routine may be needed to disable the clock to a selected region of the PLD. While the clock is halted, the configuration and state data may be read from the device. A routine that resumes the clock may also be implemented.In some embodiments standard software tools (e.g., diff, tar, cvs) can be used on the files and directories of the VFS to perform debug functions like check pointing, interrogation, archive and roll-back of the PLD state.

[0057] In yet another embodiment, the VFS may be configured to mount (the mount function being provided by the host operating system) a bitstream file rather than manipulating the data of an actual FPGA. In this embodiment, the VFS code causes the configuration controller driver to manipulate the data of another file as if it were an actual, live FPGA configuration memory. In this embodiment, the low level function calls of the VFS driver manipulate the

contents of a file in some other standard file system, even though to the user of the VFS the appearance is that actual configuration memory of an FPGA is being manipulated.

[0058] The bitstream file is a file that is ready for downloading to the FPGA for configuring the device. The VFS may contain low level routines that emulate the memory of the FPGA such that the bitstream data is parsed into the correct, emulated memory cells. Updates are made to the bitstream file by way of actions on the files in the hierarchy.

[0059] Those skilled in the art will appreciate that various alternative computing arrangements would be suitable for hosting the processes of the different embodiments of the present invention. In addition, the processes may be provided via a variety of computer-readable media or delivery channels such as magnetic or optical disks or tapes, electronic storage devices, or as application services over a network.

[0060] The present invention is believed to be applicable to a variety of systems for accessing configuration data of a PLD and has been found to be particularly applicable and beneficial in managing viewing configuration data of an FPGA. Other aspects and embodiments of the present invention will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein. It is intended that the specification and illustrated embodiments be considered as examples only, with a true scope and spirit of the invention being indicated by the following claims.